18-643 Lecture 4: Modern FPGA Programmability: PR and SoC

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today: appreciate today's FPGAs as truly dynamic and programmable devices
- Notices
  - Handout #2: lab 0, due noon, 9/8
  - Make friends, make teams, due noon, 9/8
  - Handout #3: lab 1, due noon, 9/22
- Readings
  - PR: Section 5.6, The Zynq Book
  - SoC: Ch 2, The Zynq Book
  - skim Ch 3&10, The Zynq Book
Part 1: Dynamism of Programmability

“Field (Re)programmable”

- Programmability is a very costly feature
- When we wanted FPGA to be ASIC
  - programmability avoided manufacturing NRE
  - programmability reduces design time/cost (incremental development; at speed testing; field updates, etc.)
  - BUT once programmed at power-on, FPGA is fixed
- Let’s use programmability to be more than ASIC
  - repurpose fabric over time, at large and small time scales
  - share fabric by multiple applications concurrently
Programmability can lead to performance

- **Amdahl’s Law**: $S_{\text{overall}} = 1 / ( (1-f) + f/S_f )$
- $S_{\text{F-ASIC}} > S_{\text{F-FPGA}}$ but $f_{\text{ASIC}} \neq f_{\text{FPGA}}$
- $f_{\text{FPGA}} > f_{\text{ASIC}}$ (when not perfectly app-specific)
  - more flexible design to cover a greater fraction
  - reprogram FPGA to cover different applications

[Recall: based on Joel Emer’s original comment about programmable accelerators in general]

Bitstream defines the chip

- After power up, SRAM FPGA loads bitstream from somewhere before becoming the “chip”
  - a bonus “feature” for devices that need to forget what they do

After the power up:
- control reset
- Reverse-engineering concerns ameliorated by
  - encryption
  - proprietary knowledge

Must it be always whole chip at once? [Recall]
Partial Reconfiguration (PR)

- Some parts of fabric retain their configured “personality” while other parts are reconfigured
  - e.g., keep the external bus interface from babbling while the functionality behind is changed
- The alive part can even control the reconfig.
  - e.g., load the bitstream through the bus
- Basic technology mature but usage not prevalent under the ASIC model
- Essential to FPGA as a flexible, sharable computing device

---

PR Conceptually

- Module $\text{top}()$ instantiate submodules $\text{foo}(A)$ and $\text{bar}(B)$ with interface $A$ and $B$ respectively
  - $\text{foo}(A)$ and $\text{bar}(B)$ are “blackboxes”, i.e., interface only, no internals
  - $m1() \sim m5()$ have matching interfaces, $A$ or $B$
Static and Reconfigurable Partitions

Static partition: top

reconfig. partition: instance_foo

reconfig. partition: instance_bar

Concrete Syntax (Xilinx’s approach)

module top();
    . . . .
    foo instance_foo (a1, a2, ...);
    bar instance_bar (b1, b2, ...);
    . . . .
endmodule

module foo (input a1, a2, ... output ax, ...);

    // nothing here
endmodule

module m1 (input a1, a2, ... output ax, ...);

    . . . .
   RTL body . . .
endmodule

module m2 (input a1, a2, ... output ax, ...);

    . . . .
   RTL body . . .
endmodule

module m3 (input a1, a2, ... output ax, ...);

    . . . .
   RTL body . . .
endmodule

module m4 (input a1, a2, ... output ax, ...);

    . . . .
   RTL body . . .
endmodule

module m5 (input a1, a2, ... output ax, ...);

    . . . .
   RTL body . . .
endmodule
Implementation Flow

1. Designate `instance_foo` and `instance_bar` as *Reconfigurable Partitions* (RPs)
2. Bind `instance_foo` and `instance_bar` to their most resource-demanding variants (e.g., m2 and m4)
3. Floorplanning
   - draw bounding boxes for RPs
   - reserve enough resource for largest variant
   - place partition interface pins
4. Place-and-route full design with m2 and m4 in RPs
   - extract partial designs of just m2 and m4 RPs
   - extract static partition (full design with blank RPs)

What this looks like . . .

[Vivado Implementation Screenshot]
Implementation Flow (continued)

(Re-start from static partition and locked floorplan)
5. Bind `instance_foo` and `instance_bar` to remaining variants (e.g., `m1` and `m5`)
6. Place and route new full design
7. Extract partial designs of just `m1` and `m5` RPs
   . . . repeat for `m3` . . .

End Result
- 1 full-design bitstream (including `m2` and `m4`)
- partial design bitstreams individually for `m1`~`m5`

Yes, this flow is a little clunky

At Run Time
- Power up with full-design bitstream
- Partial bitstreams in DRAM or flash memory
- Configuration API driven by ARM or fabric
  - reconfig. time depend on size, as low as msec
  - handshake signals to pause/start partition interface
Use Case: Vision Processing Pipeline

- form desired pipelines by loading RP from bitstream collection
- re/configure a new pipeline while other pipelines uninterrupted
- time-multiplex more pipelines than physical resources

Looking Forward: Multitasking a Fabric?

- Support multi-tasking workloads with their respective acceleration modules
- Schedule FPGA fabric like CPU cycles and memory
  - keep multiple active modules in fabric
  - schedule tasks to in-fabric modules
  - schedule module replacements as task load changes
    Optimize throughput vs. latency vs. energy, etc.
- Time-multiplex fabric when required modules exceeds fabric capacity
(Over) Idealization of the Problem

- Scheduling and placement as geometric bin-packing problem
  - FPGA fabric divided into all-equivalent tiles
  - module characterized by only the arrangement of tiles consumed; no consideration for I/O
  - tasks utilize modules in quantized time slots and no state remembered across invocations

Fabric has fine-structures
Practical Technology Constraints

- Number and size of RPs fixed apriori
  - too few/too large: internal fragmentation
  - too many/too small: external fragmentation
- Not all RPs are equal—even if same interface and shape
  - a module needs a different bitstream for each partition it goes into
  - build and store upto MxN bitstreams for N partitions and M modules
- Modules need external I/O and share finite external I/O bandwidth

Security and virtualization also key open issues

Part 2: System-on-Chip FPGA
SoC is the Natural Course of Scaling

- There was a time when even a “component” is a “chip set”
- Eventually, entire component fit in one chip
  \[\Rightarrow\] system \(\equiv\) board-level connection of chips
  e.g. CPU, cache, DRAM, controller, I/O, ...
- Later when more could fit on one chip
  - integrate for performance (CPU+cache)
  - integrate for cost/reliability (reducing parts count)
- Still more and more (including analog)
  Your “system” is a component in another system

SoC is IP-Based Chip Design

- Complexity wall
  - designer productivity grows slower than Moore’s Law on logic capacity
  - diminishing return on scaling design team size
  \[\Rightarrow\] must stop designing individual transistors
- A chip design as a collection of IPs
  - each IP fits in a manageable design complexity
  - IP integration fits in a manageable design complexity

Analogous to board-level chip integration in the 80s/90s
SoC is Systematic Interconnect

- More IPs, more elaborate IPs ⇒ intractable to design wires at bit- and cycle-granularity
- On-chip interconnect standards (e.g. AMBA)
  - familiar “bus” abstraction (addr/data, master/slave)
  - some even with cache-coherence
  - point-to-point signaling underneath, even full-fledge network-on-chip
- Plug-and-play integration of interface-compatible IPs

  Analogous to expansion bus and I/O cards on a motherboard

SoC is Heterogeneous Computing

- SoC integrates diverse specialized functionalities
- When power is more precious than transistor count, heterogeneous computing trades off
  - relatively small cost of unused transistors (gated off) at any given time
  versus
  - large power and performance gains from having the right tool at the right time
SoC is HW/SW Co-Design

- SoC mixes HW and SW computing substrates
- An application is partitioned for mapping to
  - HW: everything SW is not good enough for
  - SW: everything else
- SW is the heart and soul
  - in control of HW
  - enables product differentiation
- SW can be harder than HW (Is this surprising?)
  - embodying most of the complexity
  - dominating development time/effort

Meet the Zynq SoC
Yeap, it is a system (where is the FPGA?)

![Diagram](Figure 3-2, Zynq-7000 All Programmable SoC Technical Reference Manual)

Concept: Bus and Transactions

- All devices in system connected by a “bus”
  - masters: devices who initiate transactions
  - slaves: devices who respond to transactions
- Transaction based on a memory-like paradigm
  - “address”, “data”, “reading vs. writing”
  - master issues read/write transaction to an address
  - each slave is assigned an address range to respond in a “memory-like” way, i.e., returning read-data or accepting write-data

AXI is the standard interface in Zynq
Concept: Split-Phase Bus Transactions

- Asynchronous request/response queues
  - multiple outstanding transactions in flight
  - in-order or out-of-order (need tags)
- No centralized arbitration; push request when not full
- No broadcast; only addressed slave sees transaction

Concept: Memory Mapped I/O

- Think of normal ld/st as how processor “communicates” with memory
  - ld/st address identifies a specific memory location
  - ld/st data conveys information
- Can communicate with devices the same way
  - assign an address to register of external device
  - ld/st from the “mmap” address means reading/writing the register
  - BUT remember, it is not memory,
    - additional side-effects
    - not idempotent
PS/PL Interface Options

Zynq terminology:
PL=programmable logic
PS=processing system

System “bus”

something for every occasion, boon or bane?

[Fig 3-2, Zynq-7000 All Programmable SoC Technical Reference Manual]

Fabric Module as AXI Slave

- ARM core issues ld/st instructions to addresses corresponding to “mmapped” AXI device registers
  aka programmed I/O or PIO
- Nothing is simpler
- Very slow (latency and bandwidth)
- Very high overhead
  - ARM core blocks until ld response returns
  - many 10s of cycles

Best for infrequent, simple manipulation of control/status registers
Fabric Module as AXI Master

1. Fabric can also issue mmap read/write as master
2. AXI HP
   - dedicated 64-bit DRAM read/write interfaces
     fastest paths to DRAM (latency and bandwidth)
   - no cache coherence
     • if data shared, ARM core must flush cache before handing off
     • major performance hiccup from (1) flush operation and (2) cold-cache restart
     • best for fabric-only data, DRAM-only data, or very coarse-grained sharing of large data blocks

Fabric Module as ACP Master (cont.)

3. “Accelerator Coherence Port”
   - fabric issues memory read/write requests through ARM cores’ cache coherence domain
   - shortest latency on cache hits
     • ARM core could even help by prefetching
     • if not careful, ARM cores and fabric could also interfere through cache pollution
   - not necessarily best bandwidth (only one port)

Best for fine-grained data sharing between ARM cores and fabric
**DMA Controller**

- AXI-slave programming interface
  - programmable from ARM core and fabric
  - source and dest regions given as \(<\text{base}, \text{size}>\)
  - source and dest could be memory (cache coherent) or mmapped regions (e.g., ARM core scratch-pad or mmapped accelerator interface)
- Need to move large blocks to “amortize” DMA setup costs (PIO writes)
- Corollary: need to start moving well ahead of use

Best for predictable, large block exchanges

---

**Zynq Book Tutorial 1:**

SoC’ness is inherent to Vivado/Zynq thinking

- Processor-first development
- All peripherals AXI-interfaced
- Extensive library of infrastructural and functionality IPs
- You could add custom-built soft IPs
  - compile from C and Simulink, or hack RTL
  - adhere to pre-defined interface schemes
- You could program pins-and-gates too (ISE) if you enjoy “grunt work”
Parting Thoughts

• Put partial reconfiguration to use
  – works quite well (since 2002)
  – gives FPGA a distinct leg-up from ASIC
• SoC’ness complements FPGA’ness
  – hardware performance that is flexible
  – fast design turnaround (time-to-market)
  – low NRE investments
  – in-the-field update/upgrades
• FPGA “architecture” still in flux; don’t let what you see today shoehorn your imagination

The ZedBoard Kit

• ECE is loaning to you
  – 1x Avnet ZedBoard 7020 baseboard
  – 1x 12 V power supply
  – 1x 4-GB SD Card (don’t need to use it right away)
  – 2x Micro-USB cable

• You will treat it like your grade depended on it
• You will return all of it in perfect condition, or else
Looking Ahead

- Lab 1 (wk3/4): get cozy with Vivado
  - most important: learn logic analyzer and eclipse debugger
- Lab 2 (wk5/6): meet Vivado HLS
  - most important: decide if you would use it
- Lab 3 (wk7/8): hands-on with AFU
  - most important: have confidence it can work
- Project . . .